**Class Project 1- Graph Analytics**

Introduction to Big Data & Analytics

CSCI 6444

Group 13- Amish Raj, Arham Ashfaq  
Submitted to Professor Stephen Kaisler

**1. Dataset Description**

The dataset is a graph of links of opinions from the SOC-E website. In this dataset, the names have been removed and replaced by numbers. It is a medium sized dataset where each row has the format <node1>, <node2>, #edges and there are approximately 10 million nodes. Since for this dataset, #edges is 1 for each row, we will be removing the #edges column after loading the dataset as a data frame.

**2. Install the igraph package from one of the CRAN mirrors. Determine how to create a graph and plot. Show the plot in your report.**

We first install **igraph** package from one of the CRAN mirrors as described in the problem statement. We can do so by typing **“install.packages(igraph)”** in the console and then **“library(igraph)”** to use it in our project. We get an output of the form below-

Graphical user interface, text

Description automatically generatedGraphical user interface, application

Description automatically generated

Next, we **load the dataset** (keeping the **soc-Epinions1\_adj.tsv** file in the same working directory), **convert it to a matrix**, and **extract two vectors** v1 and v2. By doing this, we get the vertices in the form of vectors. We do not use the third column since that denotes the #edges which is 1 as we know for all rows.

Once we have the vectors, we can create a data frame and then a graph from it using igraph’s **graph\_from\_data\_frame()** function. In this manner, we successfully create our graph.

The code snippet for these steps is given below-

Graphical user interface, text, application

Description automatically generatedTable

Description automatically generated

As we can see in the code snippet above, once our **graph “g”** is ready, we proceed to plot the graph. We will be using the notation “g” for our graph later on as well.

Initially, the generated plot from simply running plot(g) had a very high execution time, and the resulting plot was extremely convoluted. Therefore, we reduce vertex size to 5 and edge arrow size to 0.1. This gives us a much more comprehensible graph as shown below.

Shape

Description automatically generated

However, in order to understand the plot further and simplify the graph representation, we compiled the plots of the first 100, 1000, 10000, and the last 100, 1000 and 10000 nodes. These are represented as follows-

Chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedShape

Description automatically generated with low confidenceChart, scatter chart

Description automatically generatedA picture containing shape

Description automatically generated

**3. Apply the functions shown in the Introduction to Graph Analytics document on Blackboard on the graph generated from the data set.**

Following the rubric, we apply the functions in the Introduction to Graph Analytics document and describe the results as screenshots, along with a description what the results can tell us about the problem domain.

Some of the outputs are too large and hence, most rows get omitted during printout or in the screenshot.

1. **V(g): Vertices of a Graph**  
   Table

   Description automatically generatedThe **V(g)** function returns the vertices of a graph, and we see that in graph g we have 75879 vertices. Since names have been removed and replaced by numbers in this dataset, that is what we see as labels for the nodes here.
2. **E(g): Edges of a Graph  
   Table

   Description automatically generated**The **E(g)** function is like V(g) except that it returns the edges of a graph, and in our graph g we have 811480 edges. This gives us more information about the dataset.
3. **get.adjacency()  
   Calendar

   Description automatically generated with low confidence**

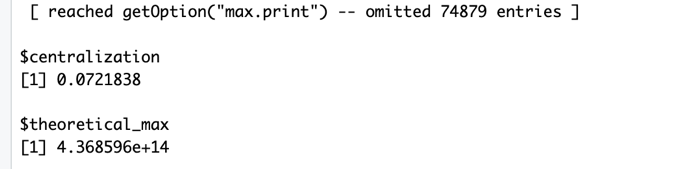
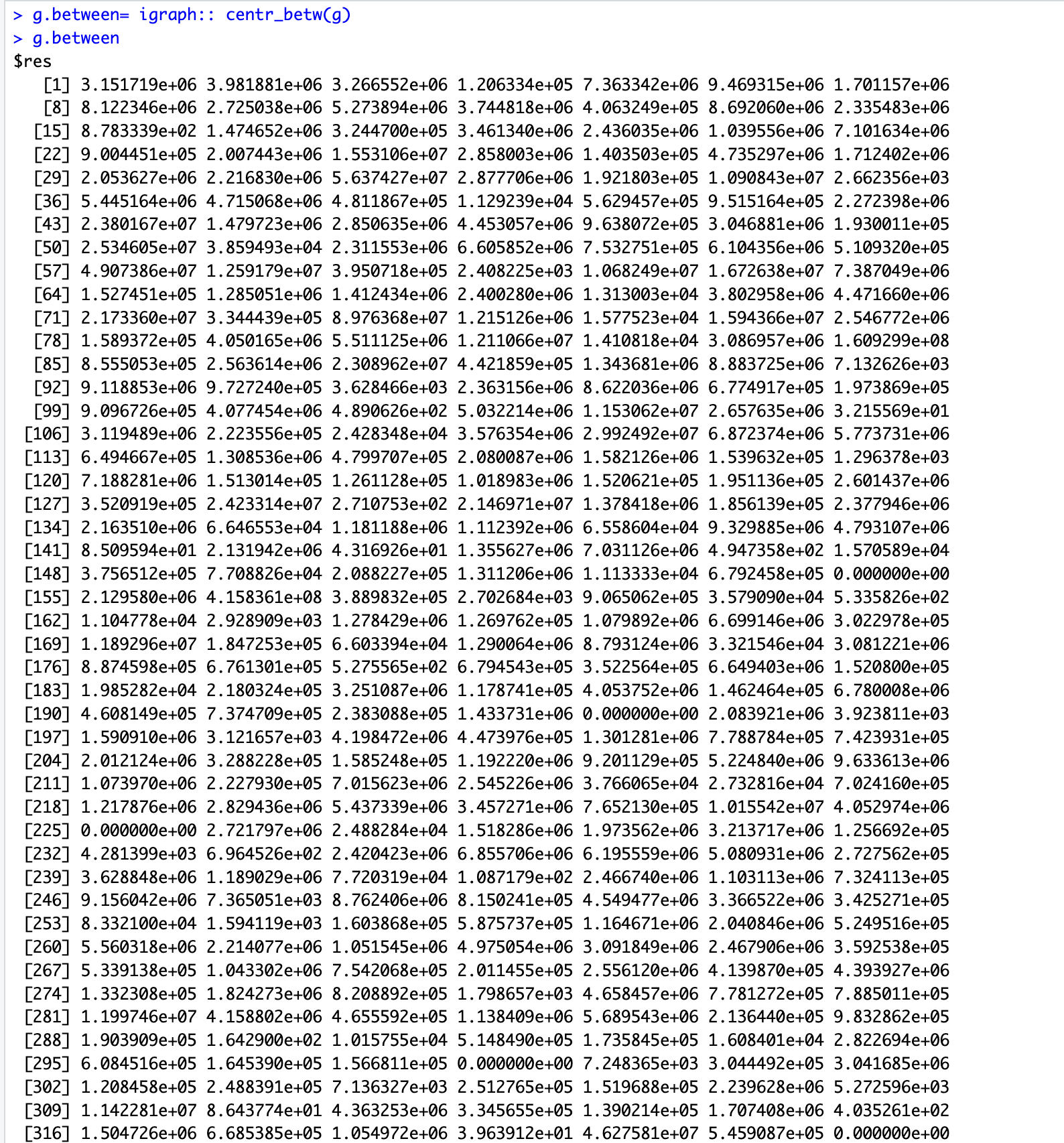
The **get.adjacency()** function returns the adjacency matrix for a graph. Here we get our graph g’s 758789 x 75879 adjacency matrix. The adjacency matrix gives us an idea about the density of our graph and even helps us identify any isolated vertices.   
For example, we can notice that the node 875 is relatively less well connected to other nodes as opposed to node 337. In this manner, we can use this information to manipulate the graph for further analysis.

1. **gden()  
   A picture containing graphical user interface

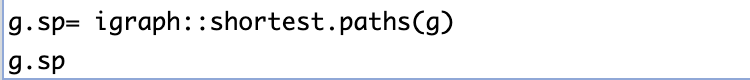
   Description automatically generated**The **gden()** function computes the density of a graph, considering and adjusting for the type of graph. Here, we can see the density of our graph is 0.1569413, which means that our graph is relatively sparse i.e., the number of edges that exist are very little compared to all the edges that could exist.
2. **edge\_density()  
   A picture containing graphical user interface

   Description automatically generated**The density of a graph is defined as the ratio of the number of edges in the graph to the maximum number of edges that could exist in the graph, which is what the **edge\_density()** function computes. We can see that even by allowing loops in the graph, the density is almost the same, which is an indicator that this graph probably does not contain loops (and therefore, might be simple graph).
3. **degree()  
   Text, calendar

   Description automatically generated with medium confidence**The **degree()** function is defined in several packages however here, we use the one defined in the igraph package. It returns the number of adjacent edges for each vertex.   
     
   From the results, we get information about the general structure and density of the graph, which we can infer by looking at the degrees of the different vertices. Let us visualize the degrees of the nodes of the graph using a histogram.  
     
   **Graphical user interface

   Description automatically generated with medium confidence**From this we can infer that most nodes have a degree between 0 and 500, and the rest have the degrees between 500-1000.
4. **centr\_betw() : Betweenness Centrality  
   **The **betweenness centrality** tells us about the number of shortest paths that pass through each vertex. In general, we can see that our graph g has relatively lower betweenness for its nodes, which means that there are fewer nodes with high betweenness (nodes that have a strong influence on the connectivity of the network).  
   This is desirable in some scenarios, however in other scenarios it might also mean that the nodes in the graph are more isolated from the rest – which might not be desirable.
5. **centr\_clo(): Closeness Centrality  
   A picture containing text

   Description automatically generatedA picture containing text

   Description automatically generated**The **centr\_clo()** function returns the closeness centrality of the various nodes of the graph.. The higher the closeness, the closer the node is to other nodes in the graph.  
   Here, we observe an average closeness centrality of 0.3, which indicates that the nodes in the graph g are fairly well connected and can reached from most other nodes. We can visually see the proof of this as well from our plots earlier.
6. **shortest.paths: Shortest Path Between Two Nodes  
     
   A screenshot of a computer

   Description automatically generated with medium confidence**The function igraph::shortest.paths(g) provides us the shortest path between any two nodes in a graph g. Here they are represented as a matrix for a shortest path from node i to node j.   
   Using this, we can locate vertices that are more central than others or find the more isolated nodes. For example, in our shortest path output above, the last path with value 41792 is less preferable than a shortest path of 7400 in the row right above it. There are some entries with over 60000 as the shortest path, and based on the use case, these observations are very useful.
7. **get.shortest.path(): Get shortest paths between vertices in a graph  
     
   Text

   Description automatically generated**The shortest paths function here helps us get the length as well as the actual shortest paths from node 5 in our graph g. This is very useful as we can gauge how well the node (5, in this case) is connected to other vertices. Here the node 5 has the shortest path length value of 3 or 4 for most vertices in the output, indicating it is relatively well connected.
8. **max\_cliques()  
   A picture containing text

   Description automatically generated**The max\_cliques() function from the igraph package helps us identify the largest cliques in a graph, along with the size of the largest clique. Here, for node 20 the largest clique is 2 with the nodes 43492 and 1310. A clique of 2 tells us that these nodes are highly connected to each other but not necessarily to the rest of the graph. A higher clique would indicate densely connected nodes.
9. **clique\_num(): Largest clique in the graphA picture containing graphical user interface

   Description automatically generated**The clique\_num() function calculates the size of the largest clique in the graph, which in the case of our graph, is 23.
10. **Simplify() and is.simple()**

**A picture containing background pattern

Description automatically generatedA picture containing text

Description automatically generated**The is.simple() method tells us if our graph is simple i.e., if our graph has no loops and multiple edges between vertices. Without performing any simplification, we notice that our graph is simple. This is verified when we apply the simplify() function on the graph after which, the is.simple() method still returns true.

1. **Detecting Structures using walktrap.community()  
   A picture containing logo

   Description automatically generated  
   **The **walktrap.community()** function tries to find densely connected subgraphs or communities in a graph through random walks since short random walks tend to stay in the same community.   
   In the plot above, we can see the various communities highlighted. Most of the subgraphs are dense and close together towards the center, whereas a few outliers exist outside the blob.

**4. Graph Simplification**

The first step we took in order to simplify our graph was to apply the simplify() function. However, as we demonstrated before, this does not give us a productive result as our graph is already simple i.e. it has no loops and multiple edges between vertices.

**A picture containing background pattern

Description automatically generated**

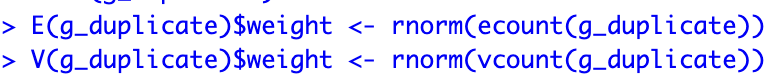
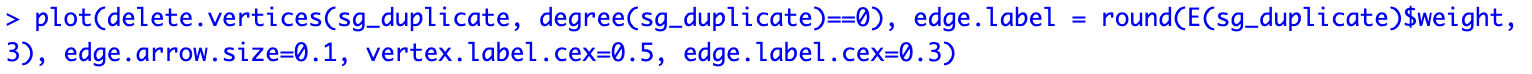
This is why, if we try to find communities in the simplified graph and plot it, we don’t see a lot of difference from the previous plot of communities.  
A picture containing shape

Description automatically generated  
So to go a step further, we thought to remove empty entries from the matrix i.e. remove isolated nodes with degree 0. However, we notice that the graph has no nodes with degree 0. A picture containing logo

Description automatically generated

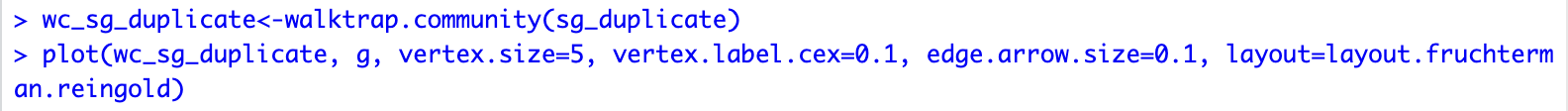
In order to perform some sort of real-world analysis on the graph, we can randomly assign weights to the vertices and the edges (as shown in the rubric on the astrocollab dataset). Upon doing that, we can fetch get a subgraph with only vertices having weight greater than 2.2 with the help of **induced.subgraph()** function.



  
  
  
A picture containing background pattern

Description automatically generated

This subgraph plot visualizes some of the highly weighted vertices in the network i.e., vertices that are more significant in the network.  
  
Note that here, we have duplicated the graph g (**g\_duplicate**) and taken a subgraph of it (**sg\_duplicate**) separate from graph g for convenience. We also remove the negative edge weights from sg\_duplicate.

When we try to find communities in the subgraph sg\_duplicate via random walks, we receive the following result-  
  


Chart, scatter chart

Description automatically generated

**5. Determining the alpha centrality, central node in the graph, longest paths, largest cliques, egos, and power centrality**

1. **Alpha Centrality:**We try to find the **alpha centrality** of our subgraph **sg\_duplicate** using igraph’s **alpha\_centrality()** function. The alpha centrality on our graph **g\_duplicate** was computationally too expensive and did not yield a result even after running for hours.  
   We see the following result for the sg\_duplicate alpha centrality:  
   Table

   Description automatically generated  
     
   We can see that the **highest alpha centrality is 46.1810604 for the node 75758**, implying that this node has a strong influence in the network.   
   Following this node, the nodes **4401**, **9202**, and **35542** have the alpha centrality **11.1791318**, **9.1115509**, and **8.8253954** respectively. We can see the other alpha centralities in descending order above.
2. **Central Node**  
   We can find the central node in two ways, either based on the sum of in and out degree or based on the node with the highest betweenness.  
     
   Based on the sum of in and out degree, we found the **central node 8886** as shown below-  
   A picture containing graphical user interface

   Description automatically generated  
     
   Based on the highest betweenness value, we found the **central node to be the same 8886** as shown below-  
   A picture containing logo

   Description automatically generated
3. **Longest Path(s)**Text

   Description automatically generated with medium confidence  
   Since the **longest path** must be in the largest connected component, we extract it using the **components()** function, and create a subgraph containing vertices within that largest connected component. We also assign the degree to each vertex attribute.  
   We can then use **DFS** to compute the largest distance from one vertex.   
     
   Our result above shows that **the longest path is 9955 from nodes 61710 and 67561**.
4. **Largest Clique(s)**Text, letter

   Description automatically generatedText, letter

   Description automatically generated  
   Here, we use the **largest\_cliques()** function to directly find the largest cliques in our graph g as well as the minimum degreerequired for all vertices in the largest clique, which is 23. It is displayed in the results above.  
   We can also use the splitting graph and binary search method to get the largest cliques.
5. **Ego(s)**We calculated the **egos** for all the vertices as shown. In our result, most of the values get omitted.  
   **A picture containing shape

   Description automatically generated**Text, letter

   Description automatically generatedText, letter

   Description automatically generated
6. **Power Centrality**We execute the **power\_centrality()** function on the subgraph (as it is not as computationally expensive as running on the original graph) and get following result: **A screenshot of a computer

   Description automatically generated with medium confidence**

**6. Discussion**

The project helps us understand how to work with datasets and generating graphs from them, especially medium to large scale datasets. We got hands experience in working with R, the igraph and sna packages, writing our own functions, and more. We gained skills related to plotting and visualizing the graphs, simplifying them, and adjusting the parameters to make a graph readable. Through the various graph analytics functions covered in this project, we gain valuable insights into the structure and relationships within the graph.

We learned about crucial measures in understanding the problem space such as alpha centrality, central node in the graph, longest paths, largest cliques, egos, and power centrality. We also learned about the process of finding communities in a graph through random walks. By understanding these measures, we can identify patterns, structures, and relationships in the data, and develop effective strategies for analyzing and visualizing large and complex networks.

To summarize, after the completion of this project, we are confident in working with R, performing graph analytics on large datasets, generating graphs, plotting them effectively, applying different functions and measures/metrics on the graph that lets us draw different inferences about the graph and the problem domain, along with simplifying graphs for better analysis.